werkexemplaar

A functional program that replaces variables by their lexical address

Maarten M Fokkinga, 16 May 1985

We present a functional program for an abstraction of the problem to replace in a program text each applied occurrence of a variable by its lexical address, that is a twotuple [n, p] where n is the lexical level of its declaration and p is the position of that declaration within that level. The fact that variables may precede their definition makes the problem interesting. The program is strongly related to an attribute grammar for the problem.

* * *

1 The problem statement

It is a well-known activety of any compiler of a block structured language to compute for each variable in a given program text its lexical address, that is a twotuple [n, p] where n is the lexical level of the declaration of that variable and p is the place of that declaration within thad level. The outermost ~~level has~~ block has level number 1 and the declarations of a block are numbered from left to right, beginning with 1. We abstract from the form a program text has, except from the nesting of blocks and the order of variables and declarations. So we consider lists like

[b, a, [B, a, b], a, B]

where capital letters represent declarations, ~~and~~ small letters represent applied occurrences of variables ≠. Notice that the first b precedes its declaration, namely the second (not the first) B. If any declaration must precede all of its applied occurrences, the program is very simple indeed.

It is the task of the requested program to transform any such given list in the desired way; the result of the program for the above list should read

[[1,2], [1,1], [B, [1, 1], [2, 1]], A, [1,1], B]

This problem has been communicated to me by S. Doaitse Swierstra.

≠: and the list structure represents the block structure

2. The program

Let us first fix some terminology and notational conventions. In the sequel the following names are used.

- n: lexical level number (a number)
- p: position (a number)
- a: address (a pair [n, p])
- v: variable (a small or capital letter)
- t: program text or bloch (a nested list over variables)
- s: syntactic environment (see below).

A syntactic environment is sometimes called symbol table or dictionary; it contains addresses for (some, not necessarily all) variables. The functions upd and find, to be defined below, serve to update environments and to retrieve information from them; they satisfy the following axiom

find (upd s v a) v' = a if v'=v
= find s v' otherwise

We choose to represent a syntactic environment as a function mapping variables onto their addresses. So we define

upd s v a = s' where s' v' = v'=v → a; s v'
find s v = s v
initS v = "undeclared"

Now we design (SASL) a function f which is a generalisation of the requested program. In addition to text t, function f has extra parameters s, n and p; and in addition to the required result t' function f yields an extra result s'. More precisely, for any call f s n p t = [t', s'] we shall see to it that

- t is ~~the~~ a part of the text to be transformed,
- s is the syntactic environment in which t is to be embedded
- n is the level number of t,
- p is the number of declarations in s on level n

and it will then hold that

- t' is the required transformation~~of~~ed t,
- s' is the environment obtained from s by updating it with the declarations from t on level n (i.e. the outermost level of t).

With this type structure of f the most creative thinking has been completed. In particular the finding of s' as an additional part of the result is crucial for an elegant solution and may be an Eureka discovery. (However, this finding is actually just a standard technique; see e.g. [Fokkinga 1985, section 13]!) Here is the defini-

tion of f and of the main function.

```
transform t = t'   where [t', s'] = f initS 1 0 t
f s n p [] = [[], s]
f s n p (x:t) =
    list x →  [x':t', s''] where [x', s'] = f s'' (n+1) 0 x
                                 [t', s''] = f s n p t
    cap x →   [x:t', s''] where [t', s''] = f s' n (p+1) t
                                s' = upd s (lowercase x) [n, p+1]
    {small x →} [a:t', s'] where [t', s'] = f s n p t
                                 a = find s' x
```

Notice that in the first clause x represents an inner block, the scope of the declarations within x does not extend outside x and therefore environment s' is not used for t. In the third clause it is environment ~~s' obtained as a by-product of processing t~~ which is used to look up x; s' is obtained as a by-product of processing t and, as may be verified from clause two, contains the declarations of the outermost level of t (in addition to what s already contains).

Exercises. 1. Change f so that it yields a triple [t', s', u] where u is a list of the undeclared variables from t.
2. Change f so that it yields [t'', s'] where t'' differs from the required t' only in that declarations are added for undeclared variables; place these declarations as much as possible at a global/local block and at the front/back of that block. 3. Treat double declarations more satisfactorily.

Notice too that f "traverses" t only once; the completion of the computation of address a (in clause three) is held up until s' has been delivered. Thus this shows ~~once more~~ that the adjectives 'one pass', 'two pass' and so on do not apply to algorithms but only to implementations, i.e. evaluations or executions of algorithms. This very same phenomenon has already been noticed earlier, see e.g. [Fokkinga 1985, section 13].

3. An attribute grammar solution

People acquainted with attribute grammars may have thought of attribute grammars right from the beginning and, even more, such grammars may have been the problem origin in the first place. We present here an attribute grammar for the problem and show the correspondence with our definition of function f.

An underlying context free grammar for blocks reads

T → empty | (T)T | VT | vT .

F instead environment s'' is used for x and s'' contains the declarations of the outermost level of t (in addition to what s already contains).

Here T, V and v are nonterminal symbols; V is to produce a capital letter and v a small letter. Nonterminal T has three inherited attributes, namely si, n, p (for: incoming environment, level number and number of declarations in si on level n), and two synthesized attributes, namely t and so (for: text produced and outcoming environment). Here are the attribute value definitions (denoting n(T') by n' and so on).

```
T → empty      t, so := [], si
T → (T) T      si', n', p' := si'', n+1, 0
               si'', n'', p'' := si, n, p
               t, so := t':t'', so''
T → V T        si', n', p' := upd si (lowercase V) [n, p+1], n, p+1
               t, so := V:t', so'
T → v T        si', n', p' := si, n, p
               t, so := a:t', so'
                     where a = find so' v
```

The correspondence with our definition of f is evident: given the text to be parsed and the inherited attributes si, n and p for that text, f simply computes the two synthesized attributes [t, so] for that text, using the same definitions as used above for the attribute values. Rea-

The resemblance is brought to an extreme if we rewrite the definition of f as follows.

```
f si n p []  = [t, so]
                  where  t, so = [], si
f si n p (x:ti) =
   list x → [t, so] where  si', n', p' = si'', n+1, 0 ;  [t', so'] = f si' n' p' x
                           si'', n'', p'' = si, n, p ;   [t'', so''] = f si'' n'' p'' ti
                           t, so = t'; t'', so''
   cap x → [t, so] where   si', n', p' = upd si (lc x) [n, p+1], n, p+1;
                                                        [t', so'] = f si'' n'' p'' ti
                           t, so = x:t', so'
{small x →} [t, so] where  si', n', p' = si, n, p ;   [t', so'] = f si' n' p' ti
                           t, so = a:t', so'
                                 where a = find so' x
```

Needless to say that we prefer the original definition of f over this attribute grammar-like formulation.

(Or to say it the other way around: is it possible to adapt the formalism of attribute grammars so that they look more like the functional program?). Re-

markably, for the functional program it is the so-called lazy evaluation strategy which controls the order of the computation steps, whereas for the attribute grammar one usually needs a separate algorithm to detect the number of passes and the order in which the attributes can be evaluated. Yet the functional program and the attribute grammar evoke the same number of computation steps (the same computation indeed); the space needed for the program may be larger than for the grammar, although its size seems to be at most as large as the complete parse tree, in general.

References

Fokkinga, M.M.: Functioneel programmeren in een vogelvlucht. Twente Univ of Techn., Netherlands, Memorandum-INF-85-4. (In Dutch)

To be inserted at the bottom of page 3

(In effect this means that environments evolve as linear lists and that find is a linear search. We leave it to the reader to represent environments as trees so that find behaves as a binary search. Assume that variables are ordered by < .)